Raksha: A Flexible Information Flow Architecture for Software Security

Michael Dalton, Hari Kannan, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{mwdalton, hkannan, kozyraki}@stanford.edu

ABSTRACT

High-level semantic vulnerabilities such as SQL injection and cross-site scripting have surpassed buffer overflows as the most prevalent security exploits. The breadth and diversity of software vulnerabilities demand new security solutions that combine the speed and practicality of hardware approaches with the flexibility and robustness of software systems.

This paper proposes Raksha, an architecture for software security based on dynamic information flow tracking (DIFT). Raksha provides three novel features that allow for a flexible hardware/software approach to security. First, it supports flexible and programmable security policies that enable software to direct hardware analysis towards a wide range of high-level and low-level attacks. Second, it supports multiple active security policies that can protect the system against concurrent attacks. Third, it supports low-overhead security handlers that allow software to correct, complement, or extend the hardware-based analysis without the overhead associated with operating system traps.

We present an FPGA prototype for Raksha that provides a full-featured Linux workstation for security analysis. Using unmodified binaries for real-world applications, we demonstrate that Raksha can detect high-level attacks such as directory traversal, command injection, SQL injection, and cross-site scripting as well as low-level attacks such as buffer overflows. We also show that low-overhead exception handling is critical for analyses such as memory corruption protection in order to address false positives that occur due to the diverse code patterns in frequently used software.

Categories and Subject Descriptors: C.0 [General]: Hardware-Software Interfaces; D.4.6 [Operating Systems:] Security & Protection – Information Flow Controls

General Terms: Security, Design, Experimentation, Performance

Keywords: Software security, Semantic Vulnerabilities, Dynamic information flow tracking, Processor architecture

1. INTRODUCTION

It is widely recognized that computer security is a critical problem with far-reaching financial and social implications [19]. Despite significant development efforts, existing security tools do not provide reliable protection against an ever-increasing set of attacks, worms, and viruses that target vulnerabilities in deployed software. Apart from memory corruption bugs such as buffer overflows, attackers are now focusing on high-level exploits such as SQL injection, command injection, cross-site scripting and directory traversals [11, 26]. Worms that target multiple vulnerabilities in an orchestrated manner are also increasingly common [1, 26].

The root of the problem is that existing approaches do not exhibit many of the desired characteristics for security techniques: *robust*: they should provide defense against vulnerabilities with few false positives or false negatives; *flexible*: they should adapt to cover evolving threats; *end-to-end*: they should be applicable to user programs, libraries, and even the operating system; *practical*: they should work with real-world code and software models (existing binaries, dynamically generated, or extensible code) without specific assumptions about compilers or libraries; and finally *fast*: they should have small impact on application performance.

Recent research has established *dynamic information flow tracking (DIFT)* [9, 17] as a promising platform for detecting a wide range of security attacks. The idea behind DIFT is to tag (taint) untrusted data and track its propagation through the system. DIFT associates a tag with every word of memory in the system. Any new data derived from untrusted data is also tagged. If tainted data is used in a potentially unsafe manner, such as executing a tagged SQL command or dereferencing a tagged pointer, a security exception is raised.

The generality of the DIFT model has led to the development of several software [4, 14, 5, 28, 13, 18, 15, 21] and hardware [24, 6, 2] implementations. Nevertheless, current DIFT systems do not exhibit all of the characteristics listed above. Software DIFT is flexible, as it can enforce arbitrary policies and adapt to different types of exploits. However, DIFT through runtime binary instrumentation leads to slowdowns ranging from 3× to 37× [21, 14]. Some software systems require access to the source code [28], while others do not work safely with multithreaded programs [21].

Hardware DIFT systems address several performance and practicality issues by performing tag propagation and checks transparently as a program executes. However, such systems use a single hardcoded security policy that targets memory corruption attacks. Hence, they cannot address high-level semantic vulnerabilities, such as SQL injection, which tend to be architecture, language, and OS-independent. Moreover, hardware DIFT systems cannot cope with binaries that violate their basic assumptions about safe/unsafe uses, or defend against attacks that evade their taint tracking rules [8]. Finally, no existing DIFT system can protect the OS code.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

This paper presents *Raksha*[1], a flexible architecture for software security using information flow tracking. Raksha provides a framework that combines the best of both hardware and software DIFT. Hardware support provides transparent, fine-grain management of security tags at low performance overhead for user code, OS code, and data that crosses multiple processes. Software provides the flexibility and robustness necessary to deal with a wide range of attacks.

Raksha introduces the following features at the architecture level. First, it provides *a flexible and programmable mechanism for specifying security policies*. The flexibility is necessary to target high-level attacks such as cross-site scripting, and to avoid the tradeoffs between false positives and false negatives due to the diversity of code patterns observed in commonly used software. Second, Raksha enables *security exceptions that run at the same privilege level and address space as the protected program*. This allows the integration of the hardware security mechanisms with additional software analyses, without incurring the performance overhead of switching to the operating system. It also makes DIFT applicable to the OS code. Finally, Raksha supports *multiple concurrently active security policies*. This allows for protection against a wide range of attacks.

In addition to defining the Raksha architecture, we have developed a prototype system by modifying an open-source SPARC processor and mapping the design onto an FPGA board. We have also modified the Linux operating system to manage Raksha's security features. The resulting system is a *full-featured Linux workstation* that can apply security policies to all memory regions (text, heap, stack) for all types of software code (dynamically generated code, self-modifying code, shared libraries, OS, and device drivers). The security framework is extensible through software, can track information flow across address spaces, and can thwart attacks employing multiple processes.

The specific contributions of this work are:

- We present the architecture and implementation of Raksha, an information flow tracking architecture for software security. Specifically, we discuss hardware support for multiple, programmable security policies and low-overhead security exceptions.
- Using a full-system prototype, we demonstrate that Raksha facilitates the integration of hardware and software security techniques that protect real-world software from a wide range of attacks. We show that Raksha is the *first* DIFT architecture to protect unmodified binaries from high-level attacks such as command injection, SQL injection, and cross-site scripting. It also provides protection against memory corruption exploits. We also show that Raksha's performance overhead is low even when coupled with intensive software analysis techniques.
- We discuss the lessons learned from applying DIFT to real-world applications in a full-system environment. Specifically, we present previously unknown false positive corner cases for both high-level and memory corruption attacks and provide mitigation strategies and directions for future research.

Overall, we show that Raksha can be the substrate for security frameworks that are robust, flexible, end-to-end, practical, and fast.

The remainder of the paper is organized as follows. Section 2 reviews related work on information flow tracking. Section 3 presents the security and performance challenges that motivated this work. Section 4 presents the Raksha architecture and Section 5 describes our full-system prototype. Section 6 evaluates Raksha's security features, while Section 7 measures performance overhead. Finally, Section 8 concludes the paper.

[1]Raksha means *protection* in Sanskrit.

2. DYNAMIC INFORMATION FLOW TRACKING

Security covers many topics including data encryption, content protection, and network trustworthiness [19]. This work focuses on detecting malicious attacks on deployed software using dynamic information flow tracking.

DIFT associates a tag with every memory word. The tag is used to taint data from untrusted sources. Most operations propagate tags from source operands to destination operands. If tagged data is used in unsafe ways, such as dereferencing a tagged pointer or executing a tagged SQL command, a security exception is raised. An important issue for DIFT-based attack detection is identifying when a tainted operand can be safely accessed without raising an exception. Depending on the type of analysis, input validation may be performed directly by the DIFT system or may be inferred from application behavior. In any case, the system must manage the following tradeoff: conservative tag clearing can lead to frequent false positive exceptions (incorrect program termination or low performance), while overly liberal tag clearing can lead to false negatives (low security).

2.1 Software-based DIFT

DIFT can be implemented by instrumenting programs at the source code level [28]. Compile-time optimizations such as on-demand allocation of tags and elimination of tagging for most local variables can significantly reduce the space and runtime overhead. Still, CPU-intensive code may slow down by 80% with memory corruption protection [28]. More importantly, this approach cannot track information flow through third party programs, binary libraries, and system calls available only in binary form. It also cannot handle inline assembly, dynamically generated code, self-modifying code, or programs written in multiple languages. Multithreaded code cannot be supported due to the potential race conditions between tag and data updates.

DIFT can be also be implemented through dynamic binary instrumentation of unmodified binaries [14, 21]. This technique is applicable to all user executables and library binaries. Software optimizations such as merging checks within and across basic blocks can reduce the overhead for CPU-intensive applications from 37× [14] to 3× [21]. However, this overhead is still too high for widespread use. This approach can neither support multithreaded code nor track information flow across multiple processes. For performance reasons, dynamic instrumentation systems support a single policy that protects against attacks on control data. It has been shown that this policy is insufficient to prevent all memory corruption cases [3].

For managed or interpreted languages, DIFT can be implemented by instrumenting the interpreter or rewriting at the bytecode level [15, 18, 13]. Such implementations have been shown to detect high-level vulnerabilities in Java and PHP code. Their main disadvantage is that any code written in other languages (e.g., JNI calls in Java) require custom wrappers for safety. Moreover, this approach cannot track information flow across multiple processes or easily deal with multithreaded executables.

2.2 Hardware-based DIFT

Hardware DIFT architectures extend each register and memory location by one tag bit. The hardware propagates and checks tags transparently as instructions execute without additional instrumentation or runtime overhead. Hardware can apply DIFT to any application, even those using self-modifying code, JIT compilation, or multithreading. All existing hardware DIFT systems focus on memory corruption attacks using a single, fixed security policy.

Minos was one of the first systems to support DIFT in hardware [6]. Its design addresses many basic issues pertaining to integration of tags in modern processors and management of tags in the OS. Minos' security policy focuses on control data attacks that overwrite return addresses or function pointers. Minos cannot protect against non-control data attacks [3].

The architecture by Suh *et al* [24] targets both control and non-control attacks by checking tags on both code and data pointer dereferences. Recognizing that real-world programs often validate their input through bounds checks, this design does not propagate the tag of an index if it is added to an untainted pointer with a pointer arithmetic instruction. This choice eliminates many false positive security exceptions but also allows for false negatives on common attacks such as return-into-libc [8]. A significant weakness is that most architectures do not have well-defined pointer arithmetic instructions. This design also introduced an efficient multi-granular mechanism for managing tag storage that reduces the memory overhead to less than 2%.

The architecture by Chen *et al* [2] provides the strongest security model for memory corruption attacks. It is similar to [24] but does not clear tags on pointer arithmetic, as there is no guarantee that the index has been validated. Instead, it clears the tag when tainted data is compared to untainted data, which is assumed to be a bounds check. As we discuss in Section 3, this approach results in both false positives and false negatives in commonly used code. Moreover, this design does not check the tag bit while fetching instructions, which allows for attacks when the code is writeable (JIT systems, virtual machines, etc) [8].

DIFT can also be used to ensure the confidentiality of sensitive data [27, 22]. RIFLE [27] proposed a system solution that tracks the flow of sensitive data in order to prevent information leaks. Apart from explicit information flow, RIFLE must also track implicit flow, such as information gleaned from branch conditions. RIFLE uses software binary rewriting to turn all implicit flows into explicit flows that can be tracked using DIFT techniques. The overall system combines this software infrastructure with a hardware DIFT implementation to track the propagation of sensitive information and prevent leaks. Infoshield [22] uses a DIFT architecture to implement information usage safety. It assumes that the program was properly written and audited and uses runtime checks to ensure that sensitive information is used only in the way defined during program development.

3. THE CASE FOR A FLEXIBLE DIFT ARCHITECTURE

The systems discussed above have demonstrated the overall potential of DIFT, and the benefits and weaknesses of hardware and software implementations. In this paper, we make the case for a flexible DIFT architecture that allows us to integrate the best of both hardware and software techniques. Specifically, we argue that hardware should provide a few key mechanisms on which software builds in order to create efficient systems that protect against a wide range of attacks.

3.1 Flexible Specification of Security Policies

Existing hardware DIFT systems use a single hardcoded policy for tag management. The policy targets memory corruption exploits such as buffer overflows. Tags are propagated from source to destination operands for all instructions. A tag is cleared when data is validated or reset. The hardware raises an exception when an instruction, a jump target, or load/store address is tagged.

A hardcoded policy is not sufficient for a robust security system. High-level attacks require tag management policies that are significantly different from memory corruption attacks. To prevent SQL injection attacks, for example, we must verify that any query passed to the SQL server does not contain tagged command characters. Unlike the rules for memory corruption that untaint tags on certain validation instructions, the SQL injection rules never untaint. The tag check rules are different as well. For SQL injection, we raise an exception to intercept the call to the SQL query execute function in order to check its input string for tainted command characters. Obviously, SQL string checks constitute a complex, high-level operation that can be done only in software, as they depend on the SQL grammar and the specifics of the database server.

Moreover, a static policy cannot cope with the diverse set of conventions in real world software that may lead to numerous false positives or negatives during a security analysis. For example, when protecting against memory corruption attacks with DIFT, we should untaint any untrusted data validated by the application. Unfortunately, validation procedures are ambiguously defined, making it difficult to describe them correctly in a single hardware policy. The architecture in [2] assumes that comparisons implement validation through bounds checking. However, we have observed numerous other validation patterns in popular software. GCC's parsing functions and glibc's `_itoa_word()` function use a modulo operation to validate hash table indices. The UID hash table in the Linux 2.6 kernel as well as glibc's `_itoa()` function validate the index by performing a logical *AND* operation because the table size is a power of 2. Even worse, comparisons are not always bounds checks. We observed a case in the traceroute program where confusing a comparison in the `free()` function for a bounds check allows for a buffer overflow attack. Overall, hardcoding the security policy can lead to one of two extremes: unwarranted validation of untrusted data that compromises security, or program termination due to false positives on legitimate uses of untrusted data.

We suggest that hardware policies for tag management should be flexible and programmable. Software should have fine-grain control over tag propagation and check rules in order to target an evolving set of attacks and to address the intricacies of real-world software.

3.2 Support for Multiple Active Policies

Existing hardware DIFT systems support a single security policy. Even if we assume a programmable policy, such systems can protect against a single attack at a time. Unfortunately, it is now common for attacks to exploit multiple vulnerabilities in a coordinated manner [1, 26]. One could consider multiplexing several policies by creating the superset of propagation and check rules for the tag bit. However, several policies define incompatible rules (e.g., string tainting vs. pointer tainting policies). Merging them would result in either a flood of spurious exceptions or a high risk of false negatives.

We suggest that DIFT architectures should support multiple concurrently active, security policies that protect against different attacks or provide mutually supportive protection against a single attack. While the exact number of active policies is still a topic of

research, our current experiments suggest that four policies are sufficient: two policies to help with high-level semantic attacks, one policy to protect against memory corruption, and one policy to assist with low-overhead security exceptions.

3.3 Low-overhead Security Exceptions

Existing DIFT architectures assume that hardware alone can fully identify unsafe uses of tagged data. Hence, tag exceptions simply trap into the OS and terminate the application. The exception overhead is not significant.

Looking forward, it is more realistic to expect that DIFT hardware will play a key role in identifying potential threats for which further software analysis is needed to detect an actual exploit. For SQL injection, for example, the hardware should track tags of inputs to the database. It will be up to software to determine if the query string contains tainted command characters, or tainted, yet harmless data. Similarly, a memory corruption policy may use software to correctly determine the code patterns that constitute input data validation. Enforcing security policies partially in software places a premium on the overhead of security exceptions. OS traps cost hundreds to thousands of cycles. Hence, even infrequent security exceptions can have a large impact on application performance. System operators should not have to choose between security and performance.

We believe that security exceptions should be handled at the user level, incurring the overhead of a function call instead of that of a full OS trap. Low-overhead security exceptions allow for software-extensible security policies without sacrificing performance. The challenge with user-level exception handling is protecting the exception handler's code and data from a potentially compromised application running at the same privilege level.

3.4 Applying DIFT to OS Code

Existing DIFT systems cannot protect the OS code primarily because the OS already runs at the highest privilege level. Hence, it is difficult to protect the security exception handler from a potentially compromised OS component. We view this as a significant shortcoming of DIFT architectures given that a successful attack against the OS can compromise the whole system. Remotely exploitable kernel vulnerabilities occur even in high-security operating systems such as OpenBSD [16]. Moreover, OS vulnerabilities often take weeks for vendors to patch after they are publicized [26].

If the system supports user-level security exceptions, we can apply DIFT to significant portions of the OS code. User-level exceptions require a mechanism to protect the handler from other code running in the same address space and privilege level. The same mechanism can protect the security handler from other OS components. We should note that a portion of the OS must still be trusted, including the security handlers themselves and the code that manages them. Still, we can check large portions of the OS, including the device drivers that are common targets of security attacks [25].

4. THE RAKSHA ARCHITECTURE

Raksha follows the general model of previous hardware DIFT systems [24, 6, 2]. All storage locations, including registers, caches, and main memory, are extended by tag bits. All ISA instructions are extended to propagate tags from input to output operands, and check tags in addition to their regular operation. Since tag operations happen transparently, Raksha can run any type of unmodified binaries without introducing runtime overheads.

4.1 Architecture Overview

Raksha differs from previous work by supporting the features discussed in Section 3.

First, it supports multiple active security policies. Specifically, each word is associated with a 4-bit tag, where each bit supports an independent security policy with separate rules for propagation and checks. As indicated by the popularity of ECC codes, 4 extra bits per 32-bit word is an acceptable overhead for additional reliability. The tag storage overhead can be reduced significantly using multi-granular approaches that exploit the common case where all words in a cache line or in a memory page are associated with the same tag [24]. The choice of four tag bits per word was motivated by the number of security policies used to protect against a diverse set of attacks with the Raksha prototype (see Section 6). Even if future experiments show that a different number of active policies is needed, the basic mechanisms described in this paper will apply.

The second difference is that Raksha's security policies are highly flexible and software-programmable. Software uses a set of policy configuration registers to describe the propagation and check rules for each tag bit. The specification format allows fine-grain control over the rules. Specifically, software can independently control the tag rules for each class of instructions and configure how tags from multiple input operands are combined. Moreover, Raksha allows software to specify custom rules for a small number of individual instructions. This enables handling of corner cases within an instruction class. For example, "*xor r1,r1,r1*" is a commonly used idiom to reset registers, especially on x86 machines. To avoid false positives while detecting memory corruption attacks, we must recognize this case and suppress tag propagation from the inputs to the output. Section 6.4 discusses complex corner cases that can be addressed using custom rules.

The third difference is that Raksha supports user-level handling of security exceptions. Hence, the exception overhead is similar to that of a function call rather than the overhead of a full OS trap. Two hardware mechanisms are necessary to support user-level exceptions handling. First, the processor has an additional *trusted mode* that is orthogonal to the conventional user and kernel mode privilege levels. Software can directly access the tags or the policy configuration registers only when trusted mode is enabled. Tag propagation and checks are also disabled when in trusted mode. Second, a hardware register provides the address for a *predefined security handler* to be invoked on a tag exception. When a tag exception is raised, the processor automatically switches to the trusted mode but remains in the same user/kernel mode and the same address space. There is no need for an additional mechanism to protect the security handler's code and data from malicious code. Raksha protects the handler using one of the four active security policies. Its code and data are tagged and a rule is specified that generates an exception if they are accessed outside of the trusted mode.

4.2 Tag Propagation and Checks

Hardware performs tag propagation and checks transparently for all instructions executed outside of trusted mode. The exact rules for tag propagation and checks are specified by a set of *tag propagation registers (TPR)* and *tag check registers (TCR)*. There is one TCR/TPR pair for each of the four security policies supported by hardware. Figures 1 and 2 present the format for the two registers as well as an example configuration for a pointer tainting analysis.

To balance flexibility and compactness, TPRs and TCRs specify rules at the granularity of *primitive operation classes*. The classes are *floating point*, *move*, *integer arithmetic*, *comparisons*, and *logical*. The move class includes register-to-register moves, loads, stores, and jumps (move to program counter). To track information

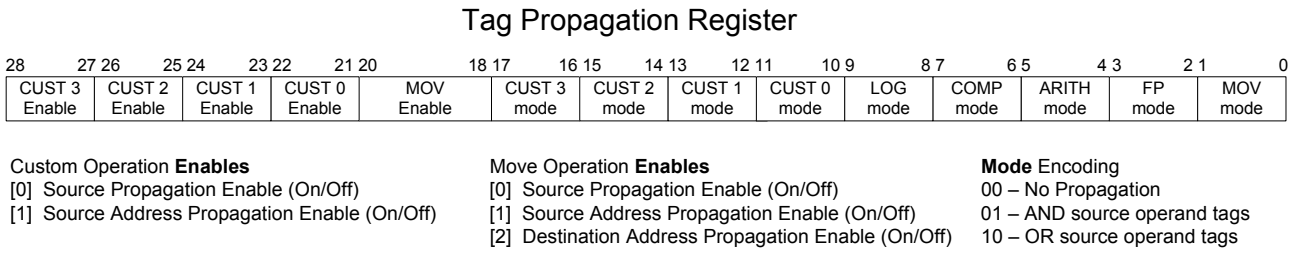

Figure 1: The format of the Tag Propagation Register. There are 4 TPRs, one per active security policy.

flow with high precision, we do not assign each ISA instruction to a single class. Instead, each instruction is decomposed into one or more primitive operations according to its semantics. For example, the `subcc` SPARC instruction is decomposed into two operations, a subtraction (arithmetic class) and a comparison that sets a condition code. As the instruction is executed, we apply the tag rules for both arithmetic and comparison operations. This approach is particularly important for ISAs that include CISC-style instructions, such as the x86. It also reflects a basic design principle of Raksha: information flow analysis tracks basic data operations, regardless of how these operations are packaged into ISA instructions. Previous DIFT systems define tag policies at the granularity of ISA instructions, which creates several opportunities for false positives and false negatives.

To handle corner cases such as register resetting with an `xor` instruction, TPRs and TCRs can also specify rules for up to four *custom operations*. As the instruction is decoded, we compare its opcode to four opcodes defined by software in the custom operation registers. If the opcode matches, we use the corresponding custom rules for propagation and checks instead of the generic rules for its primitive operation(s).

As shown in Figure 1, each TPR uses a series of two-bit fields to describe the propagation rule for each primitive class and custom operation (bits 0 to 17). Each field indicates if there is propagation from source to destination tags and if multiple source tags are combined using logical AND or OR. Bits 18 to 26 contain fields that provide source operand selection for tag propagation for move and custom operations. For move operations, we can propagate tags from the source, source address, and destination address operands. The load instruction `ld [r2], r1`, for example, considers register `r2` as the source address, and the memory location referenced by `r2` as the source.

As shown in Figure 2, each TCR uses a series of fields that specify which operands of a primitive class or custom operation should be checked for a tag exception. If a check is enabled and the tag bit of the corresponding operand is set, a security exception is raised. For most operation classes, there are three operands to consider but for moves (loads and stores) we must also consider source and destination addresses. Each TCR includes an additional operation class named *execute*. This class specifies the rule for tag checks on instruction fetches. We can choose to raise a security exception if the fetched instruction is tagged or if the program counter is tagged. The former occurs when executing tainted code, while the latter can happen when a jump instruction propagates an input tag to the program counter.

4.3 User-level Security Exceptions

A security exception occurs when a TCR-controlled tag check fails for the current instruction. Security exceptions are *precise* in Raksha. When the exception occurs, the offending instruction is not committed. Instead, exception information is saved to a special set of registers for subsequent processing (PC, failing operand, which tag policies failed, etc).

The distinguishing feature of security exceptions in Raksha is that they are processed at the user-level. When the exception occurs, the machine does not switch to the kernel mode and does not transfer control to the operating system. Instead, the machine maintains its current privilege level (user or kernel) and simply activates the trusted mode. Control is transfered to a predefined address for the security exception handler. In trusted mode, tag checks and propagation are disabled for all instructions. Moreover, software has access to the TCRs, TPRs and the registers that contain the information about the security exception. Finally, software running in the trusted mode can directly access the 4-bit tags associated with memory locations and regular registers. [2] The hardware provides extra instructions to facilitate access to this additional state when in trusted mode.

The predefined address for the exception handler is available in a special register that can be updated only while in trusted mode. At the beginning of each program, the exception handler address is initialized before control is passed to the application. The application cannot change the exception handler address because it runs in untrusted mode.

The exception handler can include arbitrary software that processes the security exception. It may summarily terminate the compromised application or simply clean up and ignore the exception. It may also perform a complex analysis to determine whether this is a false positive or try to address the security issue without terminating the code. The handler overhead depends on the complexity of the processing it performs. Since the handler executes in the same address space as the application, invoking the handler does not incur the cost of an OS trap (privilege level change, TLB flushing, etc.). The cost of invoking the security exception handler in Raksha is similar to that of a function call.

Since the exception handler and applications run at the same privilege level and in the same address space, there is a need for a mechanism that protects the handler code and data from a com-

[2]Conventional code running outside the trusted mode can implicitly operate on tags but is not explicitly aware of their existence. Hence, it cannot directly read or write these tags.

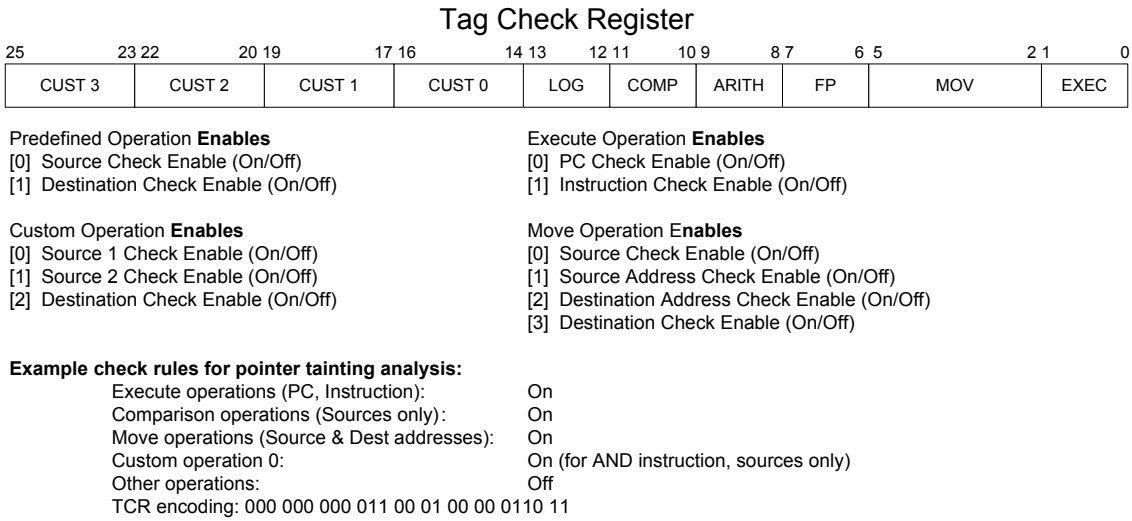

Figure 2: The format of the Tag Check Register. There are 4 TCRs, one per active security policy.

promised application. Unlike the handler, user code runs only in untrusted mode and is forbidden from using the additional instructions that manipulate special registers or directly access the 4-bit tags in memory. Still, a malicious application could overwrite the code or data belonging to the handler. To prevent this, we use one of the four security policies to sandbox the handler's data and code. We set one of the four tag bits for every memory location used by the security handler for its code or data. The TCR is configured so that any instruction fetch or data load/store to locations with this tag bit set will generate an exception. This sandboxing approach provides efficient protection without requiring different privilege levels. Hence, it can also be used to protect the trusted portion of the OS from the untrusted portion. We can also use the sandboxing mechanism (same policy) to implement the function call or system call interposition needed to detect some attacks.

4.4 Discussion

Raksha defines tag bits per 32-bit word instead of per byte. We find the overhead of per-byte tags unnecessary. Considering the way compilers allocate variables, it is extremely unlikely that two variables with dramatically different security characteristics will be packed into a single word. The one exception we found to this rule so far is that some applications construct strings by concatenating untrusted and trusted information. Infrequently, this results in a word with both trusted and untrusted bytes.

To ensure that subword accesses do not introduce false negatives, we check the tag bit for the whole word even if a subset is read. For tag propagation on subword writes, we use a control register to allow software to select a method for merging the existing tag with the new one (*and*, *or*, *overwrite*, or *preserve*). As always, it is best for hardware to use a conservative policy and rely on software analysis within the exception handler to filter out the rare false positives due to subword accesses. We would use the same approach to implement Raksha on ISAs that support unaligned accesses that span multiple words.

Raksha can be combined with any base instruction set. For a given ISA, we decompose each instruction into its primitive operations and apply the proper check and propagate rules. This is a powerful mechanism that can cover both RISC and CISC architectures. For simple instructions, hardware can perform the decomposition during instruction decoding. For the most complex CISC instructions, it is best to perform the decomposition using a micro-coding approach, as is often done for instruction decoding purposes. Raksha can handle instruction sets with condition code registers or other special registers by properly tagging these registers in the same manner as general purpose registers.

The operating system can interrupt and switch out an application that is currently in a security handler. As the OS saves/restores the process context, it also saves the trusted mode status. It must also save/store the special registers introduced by Raksha as if they were user-level registers. When the application resumes, its security handler will continue.

Like most other DIFT architectures, Raksha does not track implicit information flow since it would cause a large number of false positives. In addition, unlike information leaks, security exploits usually rely only on tainted code or data that is explicitly propagated through the system.

5. THE RAKSHA PROTOTYPE SYSTEM

To evaluate Raksha, we developed a prototype system based on the SPARC architecture. Previous DIFT systems used a functional model like Bochs to evaluate security issues and a separate performance model like Simplescalar to evaluate overhead issues with user-only code [24, 6, 2]. Instead, we use a single prototype to provide both functional and performance analysis. Hence, we can get a performance measurement for *any* real-world application that we study for security purposes. Moreover, we can use a single platform to evaluate performance and security issues related to the operating system and the interaction between multiple processes (e.g., a web server and a database).

The Raksha prototype is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [12]. We modified Leon to include the security features of Raksha and mapped the design onto an FPGA board. The resulting system is a full-featured SPARC Linux workstation.

5.1 Hardware Implementation

Figure 3 shows a simplified diagram of the Raksha hardware, focusing on the processor pipeline. Leon uses a single-issue, 7-stage pipeline. We modified its RTL code to add 4-bit tags to all

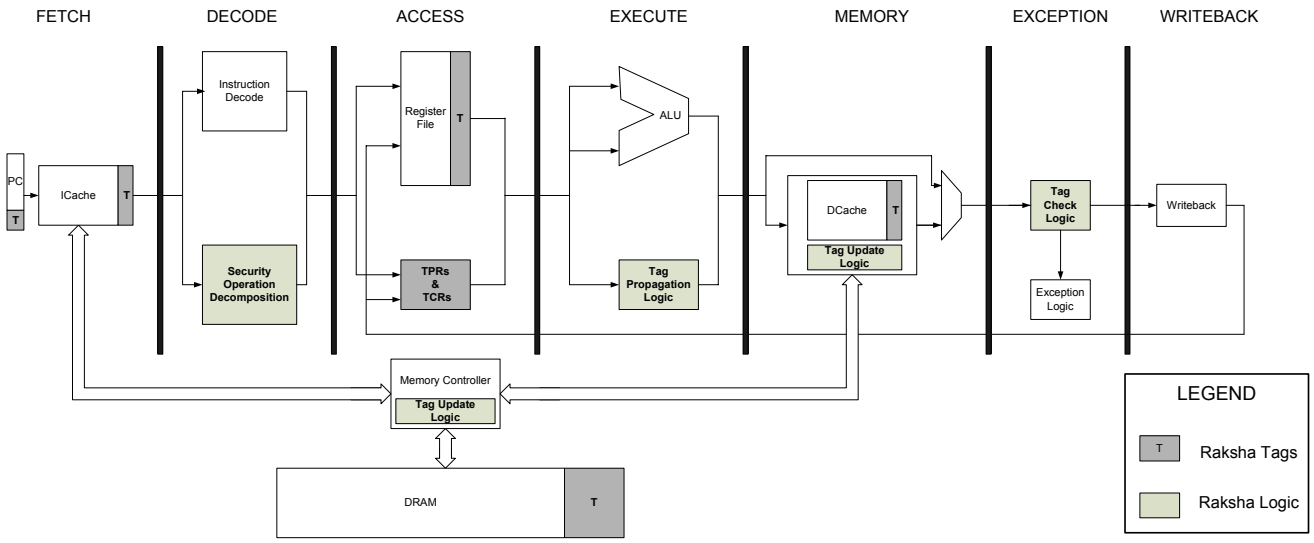

Figure 3: The Raksha version of the pipeline for the Leon SPARC V8 processor.

user-visible registers, and cache and memory locations; introduced the configuration and exception registers defined by Raksha; and added the instructions that manipulate special registers or provide direct access to tags in the trusted mode. Overall, we added 9 instructions and 16 registers to the SPARC V8 ISA. We also added support for the low-overhead security exceptions and extended all buses to accommodate tag transfers in parallel with the associated data.

The processor operates on tags as instructions flow through its pipeline as directed by the policy configuration registers (TCRs and TPRs). The Fetch stage checks the program counter tag and the tag of the instruction fetched from the I-cache. The Decode stage decomposes each instruction into its primitive operations and checks if its opcode matches any of the custom operations. The Access stage reads the tags for the source operands from the register file, including the destination operand. It also reads the TCRs and TPRs. By the end of this stage, we know the exact tag propagation and check rules to apply for this instruction. Note that the security rules applied for each of the four tag bits are independent of one another. The Execute and Memory stages propagate source tags to the destination tag in accordance with the active policies. The Exception stage performs any necessary tag checks and raises a precise security exception if needed. All state updates (registers, configuration registers, etc.) are performed in the Writeback stage. Pipeline forwarding for the tag bits is implemented similar to, and in parallel with, forwarding for regular data values.

Our current implementation of the memory system simply extends all cache lines and buses by 4 tag bits per 32-bit word. We also reserved a portion of main memory for tag storage and modified the memory controller to properly access both data and tags on cached and uncached requests. This approach introduces a 12.5% overhead in the memory system for tag storage. On a board with support for ECC DRAM, we could use the 4 bits per 32-bit word available to the ECC code to store the Raksha tags. For future versions of the prototype, we plan to implement the multi-granular tag storage approach proposed by Suh *et al* [24], where tags are allocated on demand for cache lines and memory pages that actually have tagged data.

We synthesized Raksha on the Pender GR-CPCI-XC2V Compact PCI board which contains a Xilinx XC2VP6000 FPGA. Table 1 summarizes the basic board and design statistics, including the utilization of the FPGA resources. Since Leon uses a write-through, no-write-allocate data cache, we had to modify its design to perform a read-modify-write access on the tag bits in the case of a write miss. This change and its small impact on application performance would not have been necessary had we started with a write-back cache. There was no other impact on the processor performance, as tags are processed in parallel and independently from the data in all pipeline stages.

Parameter	Specification
Pipeline depth	7 stages
Register windows	8
Instruction cache	8 KB, 2-way set associative
Data cache	32 KB, 2-way set associative
Instruction TLB	8 entries, fully-associative
Data TLB	8 entries, fully-associative
Memory bus width	64 bits
Prototype Board	GR-CPCI-XC2V board
FPGA device	XC2VP6000
Memory	512MB SDRAM DIMM
I/O	100Mb Ethernet MAC
Clock frequency	20 MHz
Block RAM utilization	22% (32 out of 144)
4-input LUT utilization	42% (28,897 out of 67,584)
Total gate count	2,405,334
Gate count increase over base Leon	7.17%

Table 1: The architectural and design parameters for the Raksha prototype.

Security features are trustworthy only if they have been thoroughly validated. Similar to other ISA extensions, the Raksha security mechanisms define a relatively narrow hardware interface that can be validated using a collection of directed and randomly generated test cases that stress individual instructions and combinations of instructions, modes, and system states. The random test generator creates arbitrary SPARC programs with randomly generated tag policies. Periodically, test programs enable the trusted mode and verify that any registers or memory locations modified since the last checkpoint have the expected tag and data values. The expected

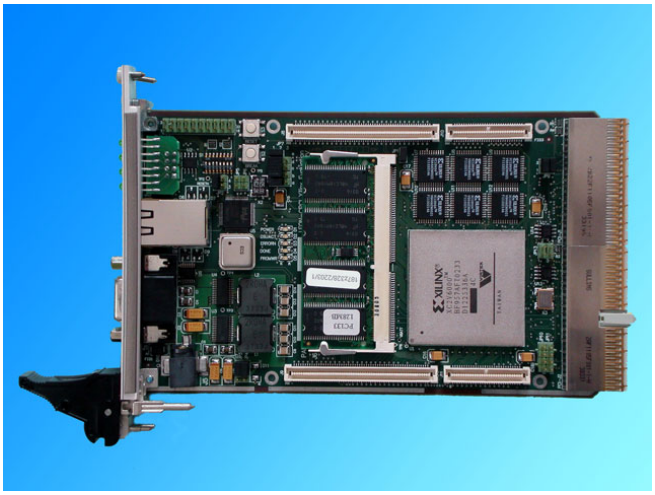

Figure 4: The GR-CPCI-XC2V board used for the prototype Raksha system.

values are generated by a simple functional-only model of Raksha for SPARC. If the validation fails, the test case halts with an error. The test case generator supports almost all SPARC V8 instructions. We have run tens of thousands of test cases on the simulated RTL using a 30-processor cluster and on the actual FPGA prototype.

5.2 Software Implementation

The Raksha prototype provides a full-fledged custom Linux distribution derived from Cross-Compiled Linux From Scratch [7]. The distribution is based on Linux kernel 2.6.11, GCC 4.0.2 and GNU C Library 2.3.6. It includes 120 software packages. Our distribution can bootstrap itself from source code and run unmodified enterprise applications such as Apache, PostgreSQL, and OpenSSH.

We have modified the Linux kernel to provide support for Raksha's security features. We ensure that the additional registers are saved and restored properly on context switches, system calls, and interrupts. Register tags must also be saved on signal delivery and SPARC register window overflows/underflows. Tags are properly copied when inter-process communication occurs, such as through pipes or when passing program arguments/environment variables to `execve`.

Security handlers are implemented as shared libraries preloaded by the dynamic linker. The OS ensures that all memory tags are initialized to zero when pages are allocated and that all processes start in trusted mode with register tags cleared. The security handler initializes the policy configuration registers and any necessary tags before disabling the trusted mode and transferring control to the application. For best performance, the basic code for invoking and returning from a security handler have been written directly in SPARC assembly. The code for any additional software analyses invoked by the security handler can be written in any programming language.

Most security analyses require that tags are properly initialized or set when receiving data from input channels. We have implemented tag initialization within the security handler using the system call interposition tag policy discussed in Section 6. For example, a SQL injection analysis may wish to tag all data from the network. The reference handler would use system call interposition on the `recv`, `recvfrom`, and `read` system calls to intercept these system calls, and taint all data returned by them.

For future versions of the Raksha software, we will focus on high-level languages for security policy specification and a centralized platform for runtime policy management. This would allow applications or operators to easily specify which policies are needed, as well as modify existing policies when new events occur (e.g., loading a plugin). We will also extend the infrastructure of security handlers to support checks on OS code, utilizing in full the hardware features of Raksha.

6. SECURITY EVALUATION

To evaluate the capabilities of Raksha's security features, we attempted a wide range of attacks on unmodified SPARC binaries for real-world applications. Raksha successfully detected both high-level attacks and memory corruption exploits on these programs. This section discusses the experiments, the policies used, and the lessons learned for two classes of attacks.

6.1 Security Experiments

Table 2 summarizes the security experiments we performed. They include attacks on basic utilities (tar, gzip, polymorph, sus), network utilities (traceroute, openssh), servers (proftpd, wu-ftpd), Web applications (Scry, Wabbit, PhpSysInfo), and search engine software (htdig). For each experiment, we list the programming language of the application, the type of attack, the DIFT analyses used for the detection, and the actual vulnerability detected by Raksha.

Unlike previous DIFT architectures, Raksha does not have a fixed security policy. The four supported policies can be set to detect a wide range of attacks. Hence, Raksha can be programmed to detect high-level attacks like SQL injection, command injection, cross-site scripting, and directory traversals, as well as conventional memory corruption and format string attacks. The correct mix of policies can be determined on a per-application basis by the system operator. For example, a Web server might select SQL injection and cross-site scripting protection, while an SSH server would probably select pointer tainting and format string protection.

To the best of our knowledge, Raksha is the *first* DIFT architecture to demonstrate detection of high-level attacks on unmodified application binaries. This is a significant result because high-level attacks now account for the majority of software exploits [26]. All prior work on high-level attack detection required access to the application source code or Java bytecode [28, 15, 18, 13]. High-level attacks are particularly challenging because they are language and OS independent. Enforcing type safety cannot protect against these semantic attacks, which makes Java and PHP code as vulnerable as C and C++.

An additional observation from Table 2 is that, by tracking information flow at the level of primitive operations, Raksha provides attack detection in a language-independent manner. The same policies can be used regardless of the application's source language. For example, htdig (C++) and PhpSysInfo (PHP) use the same cross-site scripting policy, even though one is written in a low-level, compiled language and the other in a high-level, interpreted language. Raksha can also apply its security policies across multiple collaborating programs that have been written in different programming languages.

6.2 Policies

The DIFT policies used for the security experiments are explained further in Table 3. For all but two programs, we use two concurrent security policies. We can have all the analyses in Table 2 concurrently active using the 4 tag bits available in Raksha: one for string tainting, one for pointer tainting, one for function/system call interposition, and one for the protection of the security han-

Program	Lang.	Attack	Analysis	Detected Vulnerability
gzip	C	Directory traversal	String tainting + System call interposition	`Open` file with tainted absolute path
tar	C	Directory traversal	String tainting + System call interposition	`Open` file with tainted absolute path
Wabbit	PHP	Directory traversal	String tainting + System call interposition	`Open` file with tainted pathname outside web root directory
Scry	PHP	Cross-site scripting	String tainting + System call interposition	Tainted HTML output includes $< script >$
PhpSysInfo	PHP	Cross-site scripting	String tainting + System call interposition	Tainted HTML output includes $< script >$
htdig	C++	Cross-site scripting	String tainting + System call interposition	Tainted HTML output includes $< script >$
OpenSSH	C	Command injection	String tainting + System call interposition	`execve` tainted filename
ProFTPD	C	SQL injection	String tainting + Function call interposition	Unescaped tainted SQL query
traceroute	C	Double free	Pointer tainting	Tainted data pointer dereference
polymorph	C	Buffer overflow	Pointer tainting	Tainted code pointer dereference (return address)
SUS	C	Format string bug	String tainting + Function call interposition	Tainted format string specifier in syslog
WU-FTPD	C	Format string bug	String tainting + Function call interposition	Tainted format string specifier in `vfprintf`

Table 2: The security experiments performed with the Raksha prototype.

dler. This combination allows comprehensive protection against low-level and high-level vulnerabilities.

The pointer tainting policy protects against memory corruption vulnerabilities, similar to the hardcoded protection provided by previous DIFT systems [24, 6, 2]. This policy prohibits tagged information from being used as a load address, store address, jump address, or instruction. We invoke a software security handler to recognize bounds checks through comparisons or logical AND instructions. If a bounds check is detected, we clear tags. Hence, untrusted data can be used as an array index if it has been properly validated, a common operation in real-world applications. Previous DIFT architectures simply hardcoded the policy that any comparison between a tagged and an untagged operand validates the tagged operand. This avoids the overhead of invoking the software handler through an OS trap but can lead to both false negatives and false positives (see Sections 3.1 and 6.4).

The string tainting policy is used to protect against high-level semantic attacks. It tracks untrusted data via tag propagation and allows software to check tainted arguments before sensitive function and system calls. For protection from Web vulnerabilities such as cross-site scripting, string tainting is applied both to Apache itself and to any associated modules such as PHP.

To protect the security handler from malicious attacks, we use a fault-isolation tag policy that implements sandboxing. The handler code and data are tagged, and a rule is specified that generates an exception if they are accessed outside of trusted mode. This policy ensures handler integrity even during a memory corruption attack on the application.

6.3 Lessons from High-Level Attacks

Raksha is the first DIFT system to prevent high-level attacks such as directory traversals on unmodified binaries. Raksha is well-suited to detect such high-level vulnerabilities as they tend to be precisely defined. A SQL query either has, or doesn't have, a tagged command, and Raksha security handlers can easily distinguish between safe and unsafe uses of tainted information for this class of attacks. Application and language routines for validating untrusted information do not have to be separately identified, avoiding any associated false positives and negatives.

Our experiments show that check rules for these high-level attacks must be easily customizable. For example, there is no universally accepted standard for cross-site scripting filters. A wide variety of filters are necessary to deal with the diverse set of behaviors in real-world software. Some applications HTML-encode all untrusted input; others allow input to contain a safe, restricted subset of HTML tags; and finally applications such as Bugzilla allow untrusted input to contain a restricted set of SQL commands. Because it provides programmable policies and can be extended through software, Raksha can support such customization.

Most high-level bugs can be caught at the system call layer, which has many advantages. System calls are infrequent, and interposition has minimal overhead for most workloads [10, 20]. The kernel ABI explicitly defines the semantics of each system call. Moreover, system calls provide complete mediation. If we interposed on a higher level routine, applications might evade our protection by calling directly into lower-level functions. Even though all checks are applied at the coarse granularity of system calls, the precise, fine-grain taint tracking supported by Raksha hardware is critical in order to distinguish a safe (untainted) argument to a system call from an untrusted argument that must be validated.

Prior work has shown that SQL injection can always be safely detected without false positives or negatives, so long as trusted and untrusted data can be distinguished and the SQL grammar is known [23]. Raksha does not provide perfect precision, as tags are tracked at word granularity rather than byte granularity. Strings are one of the few situations in which the same word may contain tainted and untainted bytes. Nevertheless, this has not been a significant enough issue thus far to motivate support for byte-level tags. To ensure that an application performing a byte-by-byte copy over a tainted word actually untaints the word, our string tainting policy uses merge overwrite as the tag merge mode. Our current SQL validation routine is also not as advanced as the algorithm in [23], since we scan for tainted command characters without parsing the SQL grammar. This will be addressed in future work.

Policy	Tag Initialization	Propagation Rule	Check Rule
String Tainting	Input from untrusted channel	Move (sources only) Integer Arithmetic & Logical	Analysis dependent
Pointer Tainting	Input from untrusted channel	Move (sources only) Integer Arithmetic & Logical	Move (destination & source address) Comparison (source) & Program Counter & Instruction & AND Custom operation (source)
System Call Interposition	Trap base register	Move (sources only)	Program counter
Function Call Interposition	At monitored function(s)	–	Tagged instruction
Fault Isolation	At sandboxed memory regions	–	Tagged instruction Move (destination, source)

Table 3: The tag initialization, propagation, and check rules for the security policies used in our experiments. The propagation rules identify the operation classes that propagate tags. The check rules specify the operation classes that raise exceptions on tagged operands. When needed, we identify the specific operands involved in propagation or checking.

Translation and lookup tables remain the most significant problem for web vulnerability detection using DIFT systems [8]. Our string tainting policy correctly propagates during string manipulation, copying, concatenating, etc. However, web applications may translate input from one encoding to another by indexing bytes of an untrusted string into a lookup table (e.g., convert to uppercase characters). Common glibc string conversion functions such as `atoi()` and `sprintf()` also use lookup tables. Currently, the string tainting policy will not propagate tags across such tables. If a tainted string is converted using a lookup table, then the tag bits of the resulting string will be cleared without an actual validation. Enabling move source address propagation in our string tainting policy would allow us to track tags correctly across lookup tables. However, it would also result in frequent false positives for PHP-based web applications as much of the address space becomes tainted. This is because the string propagation rules provide no mechanism for untainting a pointer except for overwriting the pointer with an untainted word. Despite these concerns, our string tainting policy did not prevent us from detecting all attacks in our experiments without any false positives. We are currently investigating better rules to address the issue of translation and lookup tables.

6.4 Lessons from Memory Corruption

Hardware and software DIFT architectures have very little information available when protecting against memory corruption vulnerabilities. Unmodified binaries do not provide bounds information and do not explicitly identify when a pointer has been validated via some sort of bounds check. Hence, DIFT systems must detect on their own when a tagged pointer should be considered safe.

However, detecting validation patterns is particularly difficult. As discussed in Section 3.1, not every bounds check is a comparison and not every comparison is a bounds check. Raksha can mitigate some of the ambiguity by using flexible security policies and perform further processing in the security handlers. To detect the case where an AND instruction is used as a bounds check, we can use a custom operation to specify a unique policy for the AND instruction. If the AND has a tagged source operand, a security exception is raised. The handler untaints the first source operand if the second source operand is untagged and is a power of 2 minus one. Previous DIFT architectures [2, 24] would not correctly identify this behavior, and would often terminate the safe program.

Unfortunately, we have also encountered other cases that cannot be resolved with hardware or software DIFT alone. Several frequently used functions in the GNU C library include tagged pointer dereferences that do not require a bounds check of any sort to be considered safe. For example, all the character conversion and classification functions (`toupper()`, `tolower()`, etc.) use 256-entry tables that can be safely indexed with tagged bytes. This is safe because the table has exactly 256 entries. If the table had 255 entries or fewer, a buffer overflow could result. Without bounds information in binaries, it is impossible for any DIFT system to resolve this corner case.

	Compare Filter		AND Filter		Combined Filter	
	Raksha	OS	Raksha	OS	Raksha	OS
bzip2	2.98x	13.20x	1.19x	1.75x	1.33x	2.80x
crafty	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
gap	1.12x	1.70x	1.00x	1.01x	1.49x	4.04x
gcc	1.01x	1.04x	1.00x	1.00x	1.00x	1.03x
gzip	1.31x	2.92x	2.39x	7.20x	2.66x	9.97x
mcf	1.00x	1.04x	1.00x	1.00x	1.00x	1.00x
parser	1.04x	1.04x	1.24x	2.28x	1.07x	1.43x
twolf	1.58x	4.19x	1.19x	1.86x	1.85x	4.48x
vpr	1.00x	1.02x	1.00x	1.00x	1.00x	1.00x

Table 4: Performance slowdown for the SPEC benchmarks with a pointer tainting analysis that filters false positives by clearing tags for select compare and AND instructions. A slowdown of 1.34x implies that the program runs 34% slower with security checks enabled.

It is reasonable to expect that additional corner cases will come up as DIFT-based memory corruption protection is used with more real-world binaries. Hence, it is important for a DIFT architecture to support software that extends or corrects the hardware checks as needed. One possible solution to the issue of false positives from difficult corner cases is to run applications on non-malicious, but tainted data, and to whitelist instructions and data that fail the memory corruption policy. The drawback to this approach is that it will only whitelist the false positives that are seen with that particular input. To support a memory corruption policy free of false positives and negatives, language or compiler help would be required. Reliable bounds information or well-defined code patterns for bounds checks would be sufficient to eliminate the above issues.

It is important to note that we observed no false positives or negatives for the code pointer protection provided by jump addresses checks and tainted instruction checks. Only the data pointer protection provided by load and store address checks have these issues.

7. PERFORMANCE EVALUATION

Hardware DIFT systems, including Raksha, perform fine-grain tag propagation and checks transparently as the application executes. Hence, they incur minimal runtime overhead compared to

program execution with security checks disabled [24, 6, 2]. The small overhead is due to tag management during program initialization, paging, and I/O events. Nevertheless, such events are rare and involve significantly higher sources of overhead compared to tag manipulation.

We focus our performance evaluation on a feature unique to Raksha - the low-overhead handlers for security exceptions. Raksha supports user-level exception handlers as a mechanism to extend and correct the hardware security analysis. As discussed in Section 6, exception overhead is not particularly important in protecting against semantic vulnerabilities. High-level attacks require software intervention only at the boundaries of certain system calls, which are infrequent events that transition to the operating system by default. On the other hand, fast software handlers can be useful in the protection against memory corruption attacks, by helping identify potential bounds-check operations and managing the tradeoff between false positives and false negatives.

Table 4 presents the slowdown experienced by various integer benchmarks from the SPEC2000 suite when software handlers are used to identify input validation cases while running the pointer tainting analysis that provides protection against memory corruption attacks. We attempted to separately filter two validation cases: comparison instructions that constitute bounds checks; and logical AND instructions with a power of two minus one that serve as bounds checks if used before indexing into a power of two sized table. The hardware raises an exception on a compare or logical AND instruction with a tagged input. The latter requires the use of a custom check policy, specific to the AND instruction. The exception handlers examines the application; if it is actually a bounds check, the handler clears the source operand tag and resumes the program execution. Clearing the operand tag avoids a false positive security exception later in the program execution. We also attempted to filter both validation cases in a combined analysis.

For every filter case, the left column in Table 4 shows the slowdown with Raksha when the software filter utilizes the low-overhead security exception. The right column measures the slowdown when the software filter is invoked through a regular OS exception. OS traps are the mechanism that previous DIFT architectures would use to invoke further software, had they recognized the need for software intervention to properly handle these corner cases.

Table 4 indicates that for programs like gcc and crafty, the overhead of software filtering is quite low for both mechanisms, as they rarely use tagged data in comparisons or logical AND instructions. On the other hand, utilities like twolf and bzip2 generate these cases more frequently. Hence, the slowdown is closely related to the overhead of the mechanism used to invoke the software filter. For gzip, Raksha's mechanism limits the overhead of compare filtering to 30%, while OS traps slow down the program by more than 2.9×. The comparison between the two techniques is similar for gzip and parser with the AND instruction filter. There are some pathological cases that run slowly on both systems. For example, bzip2 with the compare filter experiences a 3× slowdown even with user-level exceptions. On the other hand, using OS traps leads to a 13× slowdown. If a user has to choose between a 13× slowdown or program termination due to false positives, she will likely disable DIFT. While Raksha cannot eliminate all performance issues in all cases, it helps reduce the overhead of avoiding false positives and negatives in strong security policies.

Table 4 shows that the overhead for the combined filter is sometimes lower than that with one of the individual filters. This is due to the synergistic nature of the two filters. The AND filter may untag an operand that is later used multiple times in compare operations (e.g., by loading a variable from memory during each loop iteration). Another interesting observation is that the filter overheads could be reduced in some cases if, instead of just clearing the register tag, we could also clear the tag for the memory location assigned to the variable (if any).

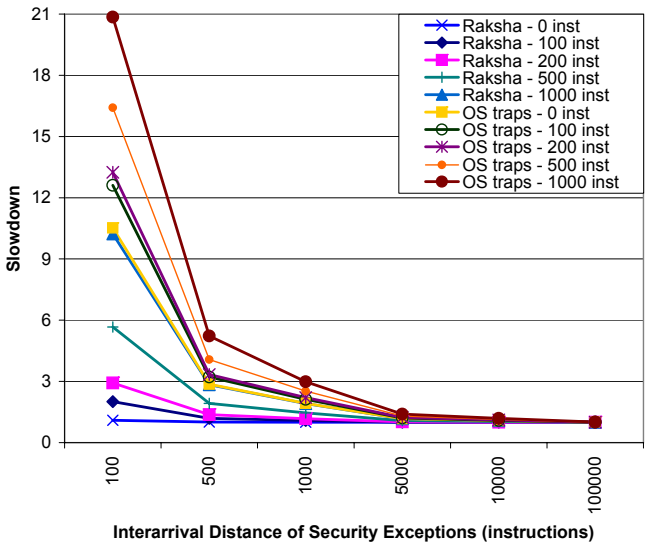

Figure 5: The performance degradation for a microbenchmark that invokes a security handler of controlled length every certain number of instructions. All numbers are normalized to a baseline case which has no tag operations.

To better understand the tradeoffs between the invocation frequency of software handlers and runtime overhead, we developed a simple microbenchmark. The microbenchmark invokes a security handler every 100 to 100,000 instructions. The duration of the handler is also controlled to be 0, 200, 500, or 1000 arithmetic instructions. This is in addition to the instructions necessary to invoke and terminate the handler. Figure 5 shows that if security exceptions are invoked less frequently than every 5,000 instructions, both user-level and OS-level exception handling are acceptable as their cost is easily amortized. On the other hand, if software is involved as often as every 1,000 or 100 instructions, user-level handlers are critical in maintaining acceptable performance levels. Low-overhead security exceptions allow software to intervene more frequently or perform more work per invocation. For reference, our software filters for the experiments in Table 4 require approximately 100 instructions per invocation.

8. CONCLUSIONS AND FUTURE WORK

We presented Raksha, a novel information flow architecture for software security. Raksha provides a framework that combines the best of both hardware and software DIFT. Hardware support provides transparent, fine-grain management of security tags at low performance overhead for user code, OS code, and data that crosses multiple processes. Software provides the flexibility and robustness necessary to deal with a wide range of attacks.

Unlike previously proposed DIFT architectures, Raksha supports flexible and programmable security policies. Software can set the security policy to provide protection against a new type of attack or identify a corner case in the interaction of existing security policies and deployed software. Raksha supports multiple active policies that allow concurrent protection against both high-level and low-level vulnerabilities. Finally, Raksha supports a user-level ex-

ception handling that allows for fast security handlers that execute in the same address space as the potentially malicious application. Overall, Raksha supports the mechanisms that allow software to correct, complement, or extend the hardware-based analysis.

We implemented a fully-featured Linux workstation as a prototype for Raksha using a synthesizable SPARC core and an FPGA board. Running real-world software on the prototype, we demonstrated that Raksha is the first DIFT architecture to detect high-level vulnerabilities such as directory traversals, command injection, SQL injection, and cross-site scripting, while providing protection again conventional memory corruption attacks. We also showed and evaluated how low overhead security handlers can be used to address the shortcomings of hardware security analysis in a performance-efficient manner.

Raksha's flexible architecture provides several opportunities for further security research. First, we will extend the security policies to support the operating system, protecting the kernel from memory corruption attacks and user/kernel pointer dereferences. We will also investigate stronger policies, such as detecting tainted system call arguments during memory corruption attacks or specifying practical tag propagation rules for lookup tables in Web applications. We are also interested in improving the infrastructure for defining and managing security policies in a manner that supports cross-process, cross-file, and whole-system information flow. Finally, we believe that the basic mechanisms in Raksha have interesting applications beyond security for services such as debugging, fault isolation, or profiling.

9. ACKNOWLEDGMENTS

We would like to thank Jiri Gaisler, Richard Pender, and Gaisler Research in general for their invaluable assistance with the prototype development. We would also like to thank David August, Suzanne Rivoire and the anonymous reviewers for their feedback on the paper. This work was supported by two Stanford Graduate Fellowships funded by Sequoia Capital and Cisco Systems, and NSF Career Award number 0546060.

10. REFERENCES

[1] CERT Coordination Center. Overview of attack trends. http://www.cert.org/archive/pdf/attack_trends.pdf, 2002.

[2] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *the Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 2005.

[3] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *the Proceedings of the 14th USENIX Securiry Symposium*, Baltimore, MD, Aug. 2005.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole system Simulation. In *the Proceedings of the 13th USENIX Security Conference*, Aug. 2004.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *the Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, Oct. 2005.

[6] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *the Proceedings of the 37th Intl. Symposium on Microarchitecture*, Portland, OR, Dec. 2004.

[7] Cross-Compiled Linux From Scratch. `http://cross-lfs.org`.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, Boston, MA, June 2006.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *ACM Communications*, 20(7), 1977.

[10] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *the Proceedings of the 11th Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2004.

[11] Imperva Inc., How Safe is it Out There: Zeroing in on the vulnerabilities of application security. `http://www.imperva.com/company/news/2004-feb-02.html`, 2004.

[12] LEON3 SPARC Processor. `http://www.gaisler.com`.

[13] B. Livshits, M. Martin, and M. S. Lam. SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities. Technical report, Stanford University, Sept. 2006.

[14] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *the Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2005.

[15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, Chiba, Japan, May 2005.

[16] OpenBSD IPv6 mbuf remote kernel buffer overflow. `http://www.securityfocus.com/archive/1/462728/30/0/threaded`, 2007.

[17] Perl taint mode. `http://www.perl.com`.

[18] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *the Proceedings of the Recent Advances in Intrusion Detection Symposium*, Seattle, WA, Sept. 2005.

[19] President's Information Technology Advisory Committee (PITAC). CyberSecurity: A Crisis of Prioritization. `http://www.nitrd.gov/pitac/reports/20050301\_cybersecurity/cybersecurity.pdf`, Feb. 2005.

[20] N. Provos. Improving Host Security with System Call Policies. In *the Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[21] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *the Proceedings of the 39th the Intl. Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[22] W. Shi, H.-H. Lee, G. Gu, L. Falk, T. Mudge, and M. Ghosh. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the 12th Intl. Symposium on High-Performance Computer Architecture*, Austin, TX, Feb. 2006.

[23] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *the Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.

[24] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *the Proceedings of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[25] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Computer Systems*, 23(1), 2005.

[26] Symantec Internet Security Threat Report, Volume X: Trends for January 06 - June 06, Sept. 2006.

[27] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *the Proceedings of the 37th Intl. Symposium on Microarchitecture*, Portland, OR, Dec. 2004.

[28] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *the Proceedings of the 15th USENIX Security Conference*, Vancouver, Canada, Aug. 2006.